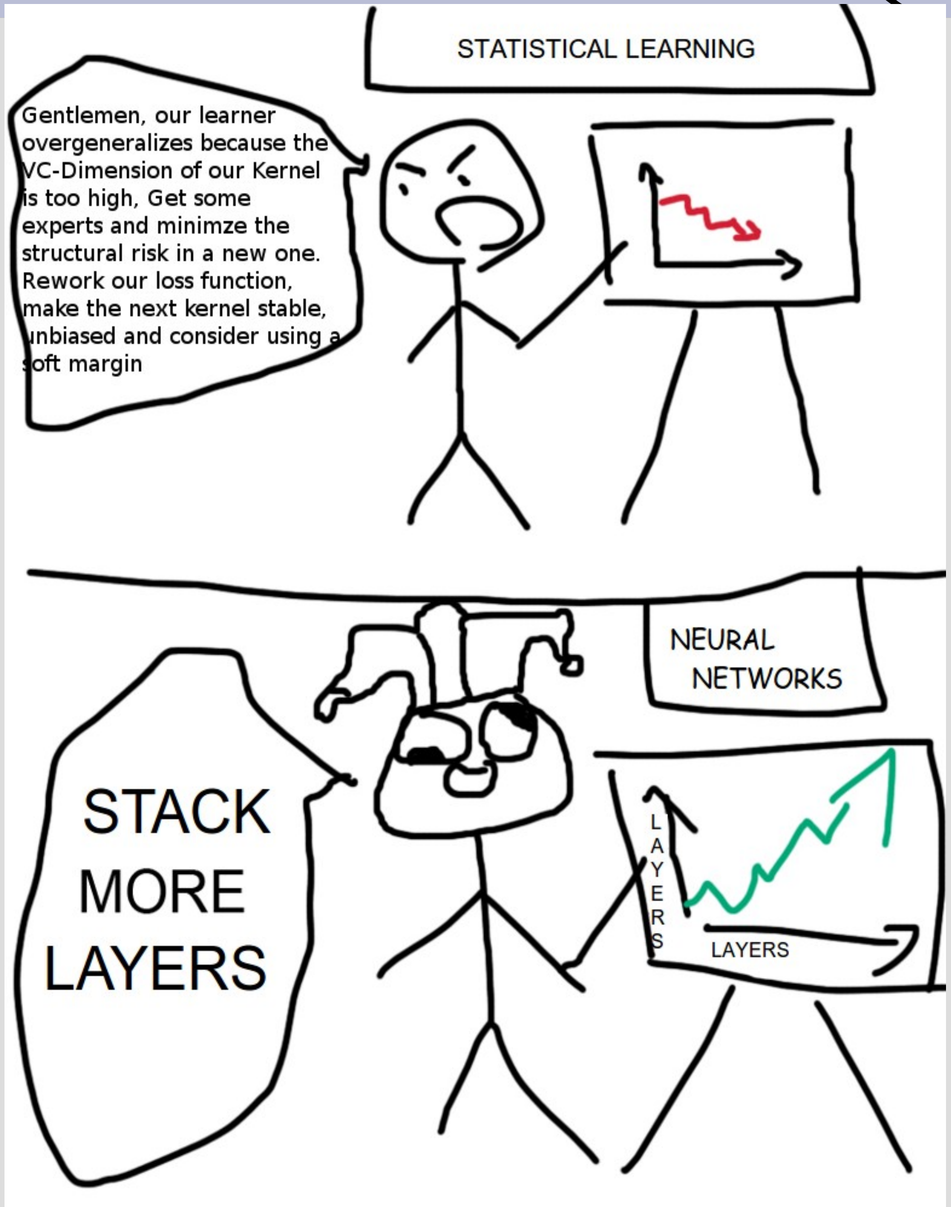
Neural networks (Ch. 12)


STATISTICAL LEARNING
Gentlemen, our learner overgeneralizes because the VC-Dimension of our Kernel is too high, Get some experts and minimze the structural risk in a new one. Rework our loss function, make the next kernel stable, unbiased and consider using a soft margin
NEURAL NETWORKS
STACK MORE LAYERS
LAYERS
LAYERS

Back-propagation

The neural network is as good as it's structure and weights on edges

Structure we will ignore (more complex), but there is an automated way to learn weights

Whenever a NN incorrectly answer a problem, the weights play a “blame game”...

- Weights that have a big impact to the wrong answer are reduced

Back-propagation

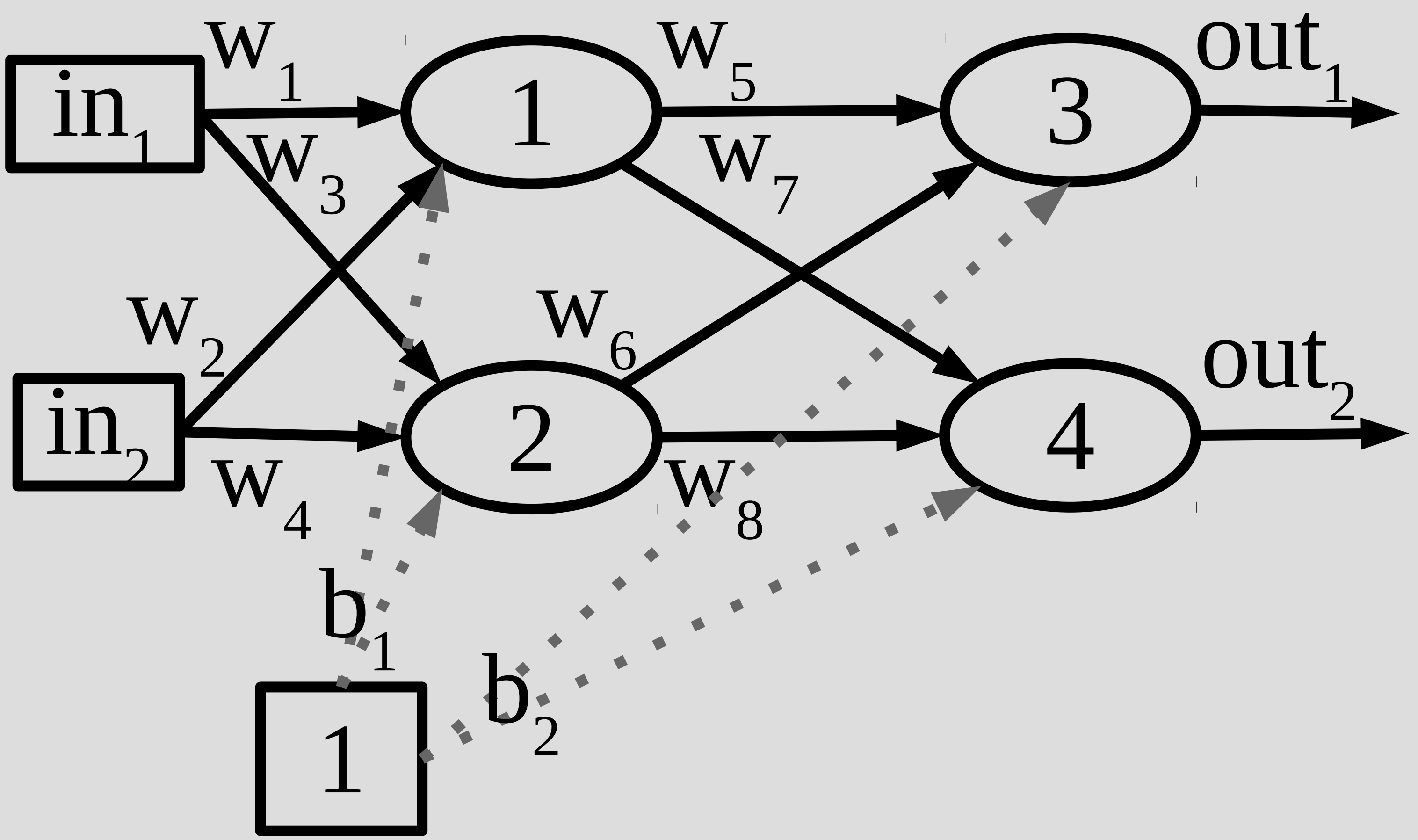
To do this blaming, we have to find how much each weight influenced the final answer

Steps:

1. Find total error
2. Find derivative of error w.r.t. weights
3. Penalize each weight by an amount proportional to this derivative

Back-propagation

Consider this example: 4 nodes, 2 layers

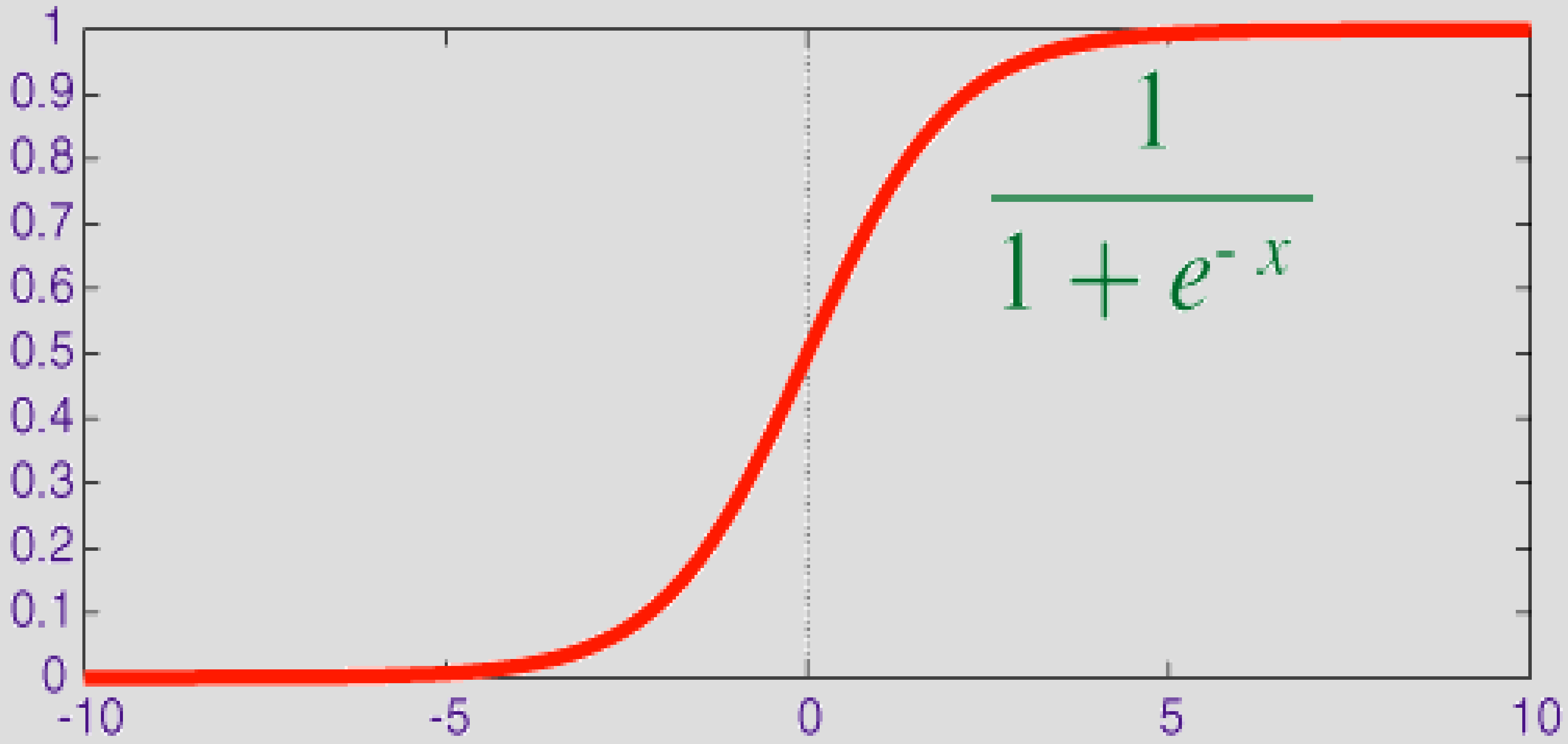


This node as a constant bias of 1

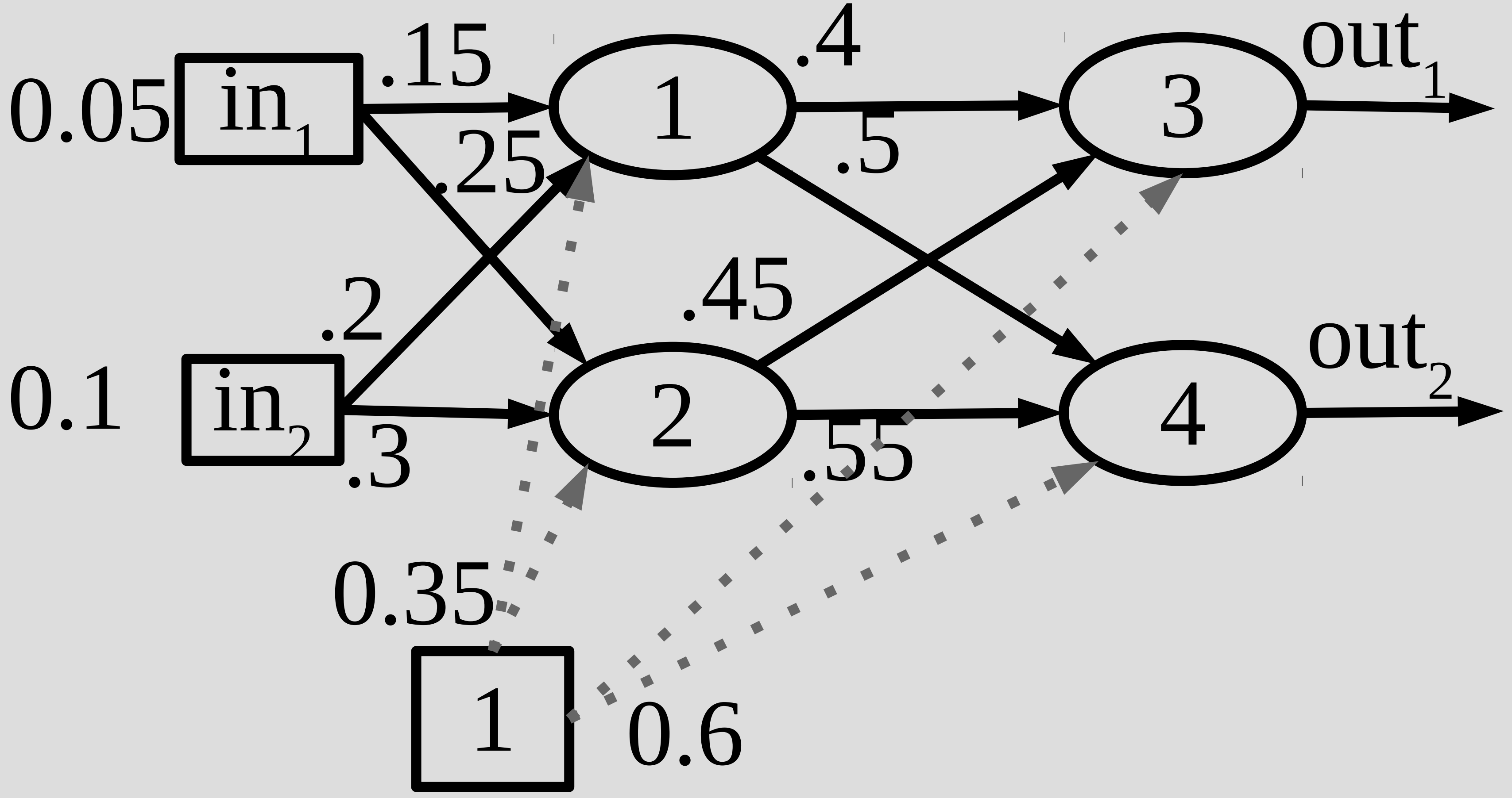
Neural network: feed-forward

One commonly used function is the sigmoid:

$$S(x) = \frac{1}{1+e^{-x}}$$

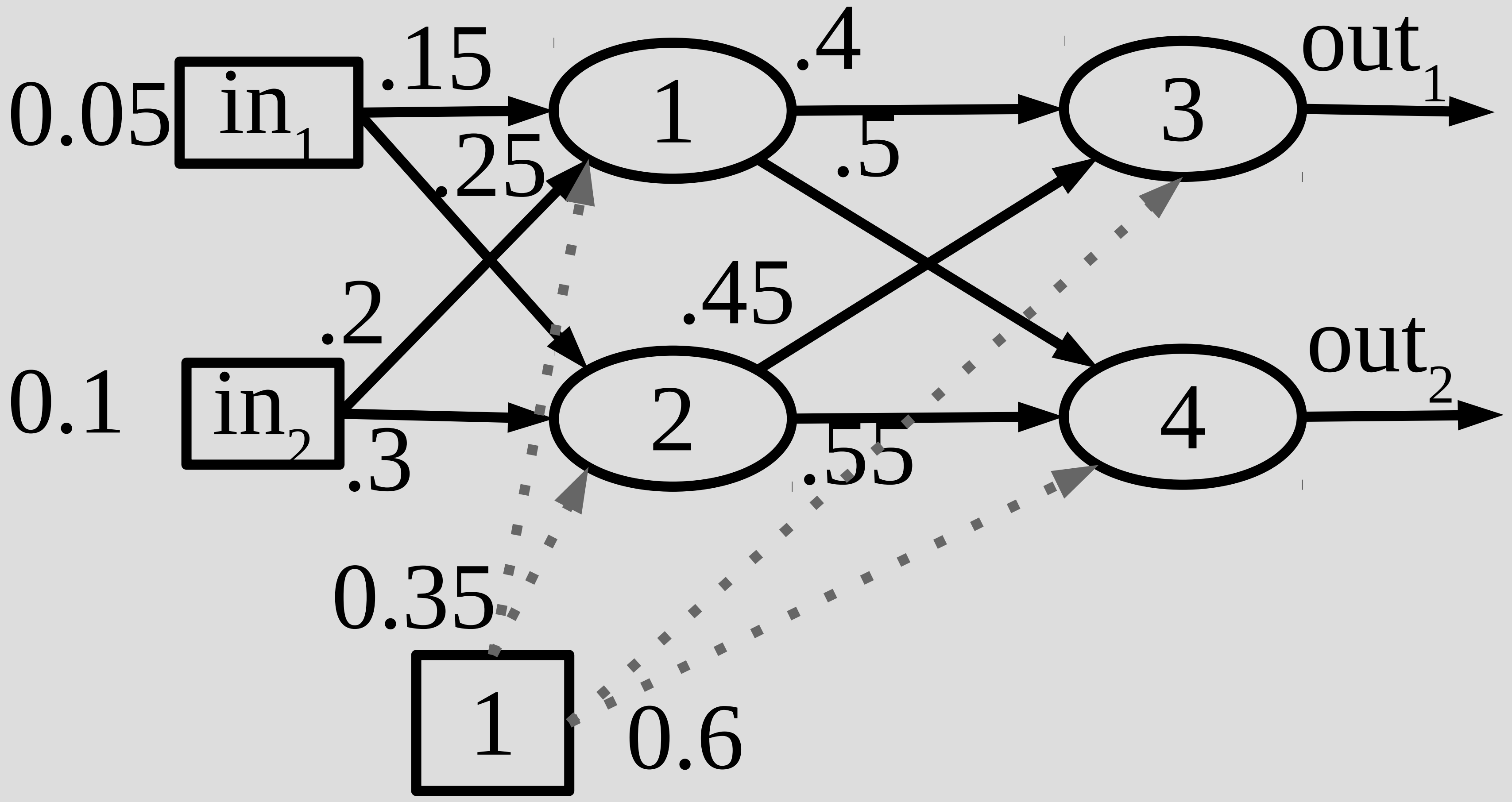


Back-propagation



Node 1: 0.15*0.05 + 0.2*0.1 +0.35 as input thus it outputs (all edges) S(0.3775)=0.59327

Back-propagation



Eventually we get: $out_1 = 0.7513$, $out_2 = 0.7729$

Suppose wanted: $out_1 = 0.01$, $out_2 = 0.99$

Back-propagation

We will define the error as: $\frac{\sum_i (correct_i - output_i)^2}{2}$
(you will see why shortly)

Suppose we want to find how much w_5 is to blame for our incorrectness

We then need to find: $\frac{\partial Error}{\partial w_5}$
Apply the chain rule:

$$\frac{\partial Error}{\partial out_1} \cdot \frac{\partial S(In(N_3))}{\partial In(N_3)} \cdot \frac{\partial In(N_3)}{\partial w_5}$$

Back-propagation

$$Error = \frac{\sum_i (correct_i - output_i)^2}{2}$$

$$\frac{\partial Error}{\partial out_1} = -(correct_1 - out_1)$$
$$= -(0.01 - 0.7513) = 0.7413$$

As $S'(x) = S(x) \cdot (1 - S(x))$

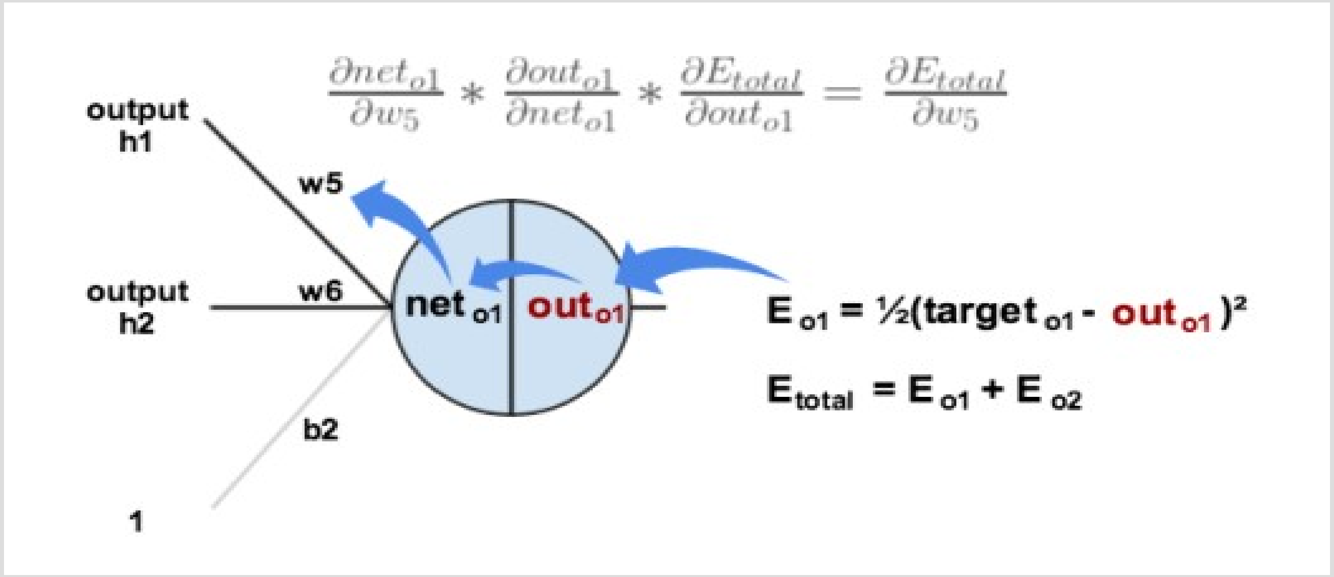
$$\frac{\partial S(In(N_3))}{\partial In(N_3)} = S(In(N_3)) \cdot (1 - S(In(N_3)))$$
$$= 0.7513 \cdot (1 - 0.7513) = 0.1868$$

$$\frac{\partial In(N_3)}{\partial w_5} = \frac{\partial w_5 \cdot Out(N_1) + w_6 \cdot Out(N_2) + b_2 \cdot 1}{\partial w_5}$$
$$= Out(N_1) = 0.5932$$

Thus, $\frac{\partial Error}{\partial w_5} = 0.7413 \cdot 0.1868 \cdot 0.5932 = 0.08217$

Back-propagation

In a picture we did this:



$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$
output h1
w5
output h2
w6
b2
1
net_{o1}
out_{o1}
$E_{o1} = ½(target_{o1} - out_{o1})^2$
$E_{total} = E_{o1} + E_{o2}$

Now that we know w5 is 0.08217 part responsible, we update the weight by:

$w_5 \leftarrow w_5 - \alpha * 0.08217 = 0.3589$ (from 0.4)

α is learning rate, set to 0.5

Back-propagation

Updating this w_5 to w_8 gives:

$w_5 = 0.3589$

$w_6 = 0.4067$

$w_7 = 0.5113$

$w_8 = 0.5614$

For other weights, you need to consider all possible ways in which they contribute

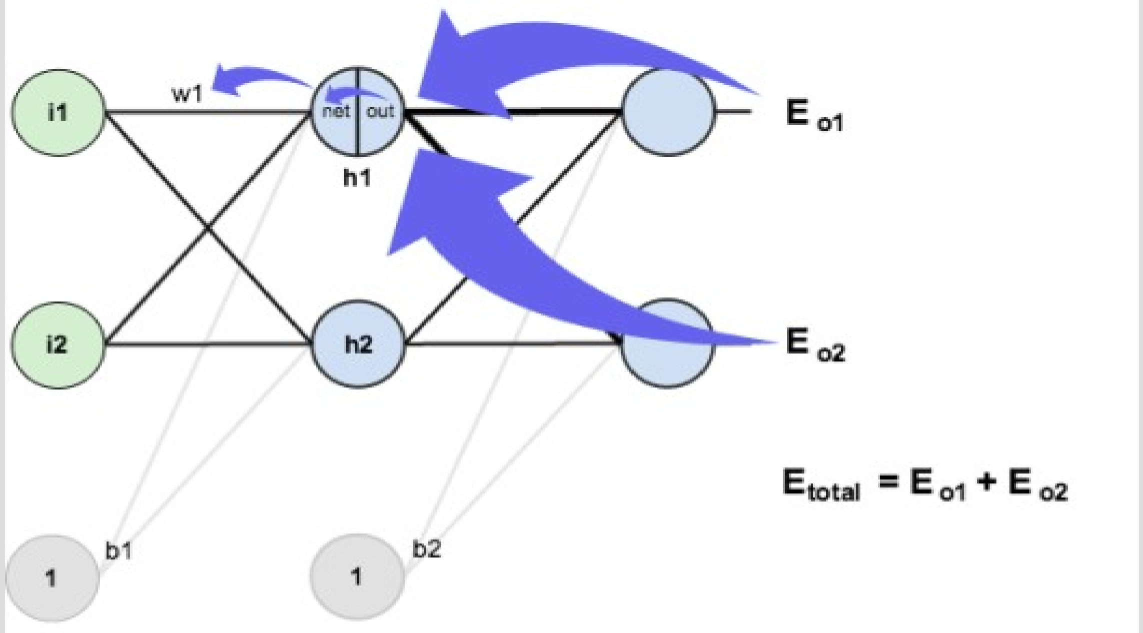
Back-propagation

For w_1 it would look like:

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\downarrow$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$



(book describes how to dynamic program this)

Back-propagation

Specifically for w_1 you would get:

$$\frac{\partial Error}{\partial S(In(N_1))} = \frac{\partial Error_1}{\partial S(In(N_1))} + \frac{\partial Error_2}{\partial S(In(N_1))}$$

$$\frac{\partial S(In(N_1))}{\partial In(N_1)} = S(In(N_1)) \cdot (1 - S(In(N_1))$$
$$= 0.5933 \cdot (1 - 0.5933) = 0.2413$$

$$\frac{\partial In(N_3)}{\partial w_5} = \frac{\partial w_1 \cdot In_1 + w_2 \cdot In_2 + b_1 \cdot 1}{\partial w_5}$$
$$= In_1 = 0.05$$

Next we have to break down the top equation...

Back-propagation

$$\frac{\partial Error}{\partial S(In(N_1))} = \frac{\partial Error_1}{\partial S(In(N_1))} + \frac{\partial Error_2}{\partial S(In(N_1))}$$

$$\frac{\partial Error_1}{\partial S(In(N_1))} = \frac{\partial Error_1}{\partial S(In(N_3))} \cdot \frac{\partial S(In(N_3))}{\partial In(N_3)} \cdot \frac{\partial In(N_3)}{\partial S(In(N_1))}$$

From before... $\frac{\partial Error_1}{\partial S(In(N_3))} \cdot \frac{\partial S(In(N_3))}{\partial In(N_3)}$
$= 0.7414 \cdot 0.1868 = 0.1385$

$$\frac{\partial In(N_3)}{\partial S(In(N_1))} = \frac{\partial w_5 \cdot S(In(N_1)) + w_6 \cdot S(In(N_2)) + b_1 \cdot 1}{\partial S(In(N_1)}$$
$= w_5 = 0.4$

Thus, $\frac{\partial Error_1}{\partial S(In(N_1))} = 0.1385 \cdot 0.4 = 0.05540$

Back-propagation

Similarly for Error$_2$ we get:

$$\frac{\partial Error}{\partial S(In(N_1))} = \frac{\partial Error_1}{\partial S(In(N_1))} + \frac{\partial Error_2}{\partial S(In(N_1))}$$
$$= 0.05540 + -0.01905 = 0.03635$$

Thus, $\frac{\partial Error}{\partial w_1} = 0.03635 \cdot 0.2413 \cdot 0.05 = 0.0004386$

Update $w_1 \leftarrow w_1 - \alpha \frac{\partial Error}{\partial w_1} = 0.15 - 0.5 \cdot 0.0004386 = 0.1498$

You might notice this is small...
This is an issue with neural networks, deeper the network the less earlier nodes update

NN examples

Despite this learning shortcoming, NN are useful in a wide range of applications:

- Reading handwriting
- Playing games
- Face detection
- Economic predictions

Neural networks can also be very powerful when combined with other techniques (genetic algorithms, search techniques, ...)

NN examples

Examples:

https://www.youtube.com/watch?v=umRdt3zGgpU

https://www.youtube.com/watch?v=qv6UVOQ0F44

https://www.youtube.com/watch?v=xcIBoPuNIiw

https://www.youtube.com/watch?v=0Str0Rdkxxo

https://www.youtube.com/watch?v=l2_CPB0uBkc

https://www.youtube.com/watch?v=0VTI1BBLydE

NN examples

AlphaGo has been in the news recently, and is also based on neural networks

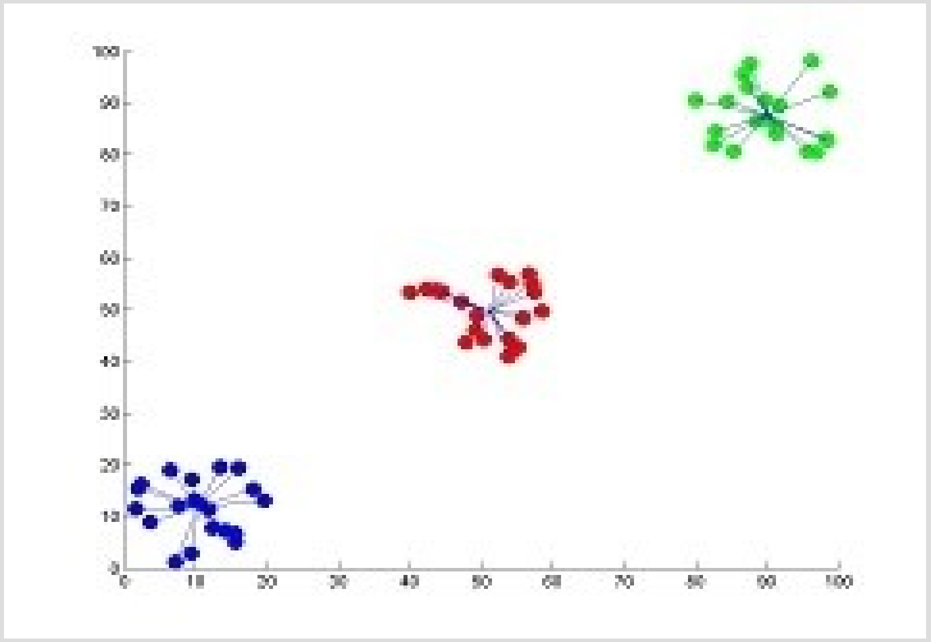
AlphaGo uses Monte-Carlo tree search guided by the neural network to prune useless parts

Often limiting Monte-Carlo in a static way reduces the effectiveness, much like mid-state evaluations can limit algorithm effectiveness

Other random topics

K-means clustering on points is finding K “central locations” that reduce the distance of each point to the nearest “central location” (summed over all points)

K=3



Other random topics

For examples like the previous one, it is easy to find which points should be “grouped together”

Once you have a group of points, you can mathematically find the best “central location”

(“center of mass” with equally massive points)

$$center_x = \frac{1}{|G_{center}|} \sum_{i \in G_{center}} x_i$$
$$center_y = \frac{1}{|G_{center}|} \sum_{i \in G_{center}} y_i$$

Other random topics

Suppose you wanted to find the best spot to put 5 “central locations” here:

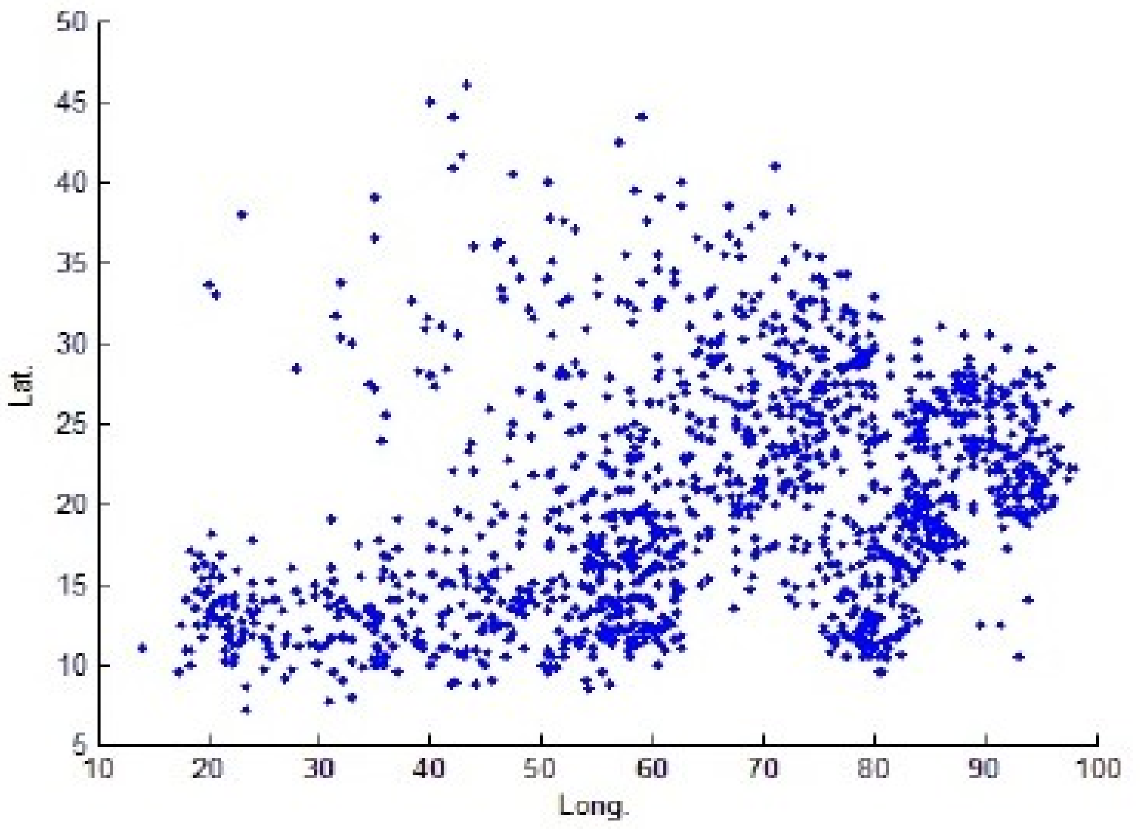


Figure 3. Starting points of all hurricane tracks

Other random topics

Suppose you wanted to find the best spot to put 5 "central locations" here:

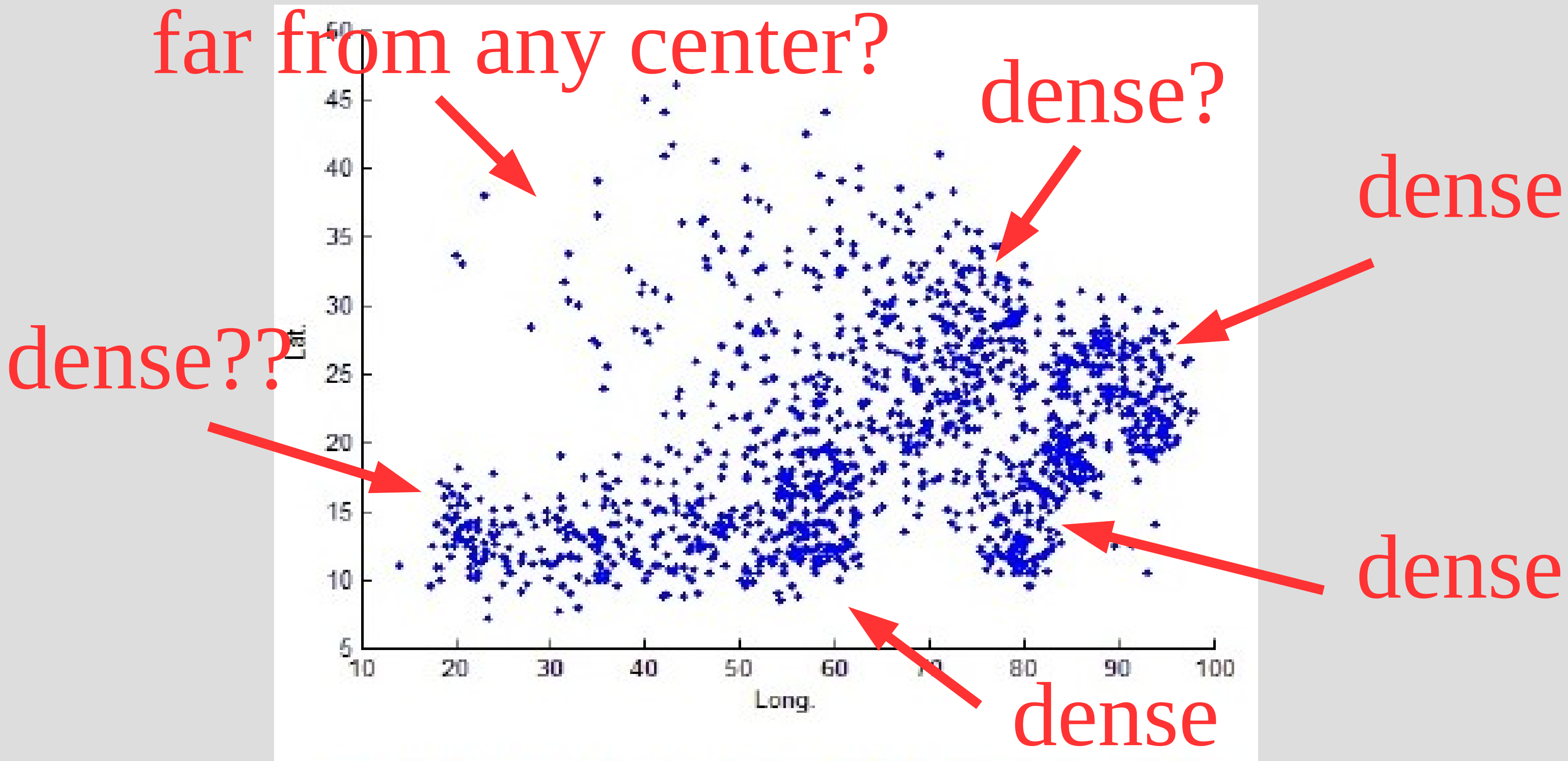


Figure 3. Starting points of all hurricane tracks

Other random topics

Turns out you can do this the other way around as well...

If you have the “central locations” (x,y) coordinates, you can find which location all points should go to (minimum distance)

Other random topics

We have a problem:

1. If we knew point groupings, we could find the best central locations

2. If we knew central locations positions, we could find point groupings

Other random topics

One common way to solve this issue when you have multiple unknowns that depend on each other is to simply guess, then try to optimize

So, initially just make random groupings

Then find the best central locations base off of the groupings

Then find the best groupings... and repeat

Other random topics

If you set up the problem correctly (and have a "well behaved" metric), this will converge

In fact, you can do this even if you have more than two unknowns

Just make one variable while fixing all others and optimize that one
... then pick a new variable to "optimize"

Other random topics

This technique actually works in a large range of settings:

K-means clustering (this)
Bayesian networks (probabilistic reasoning)
Markov Decision Processes (policy selection)
Expectation–Maximization (parameter optimization)